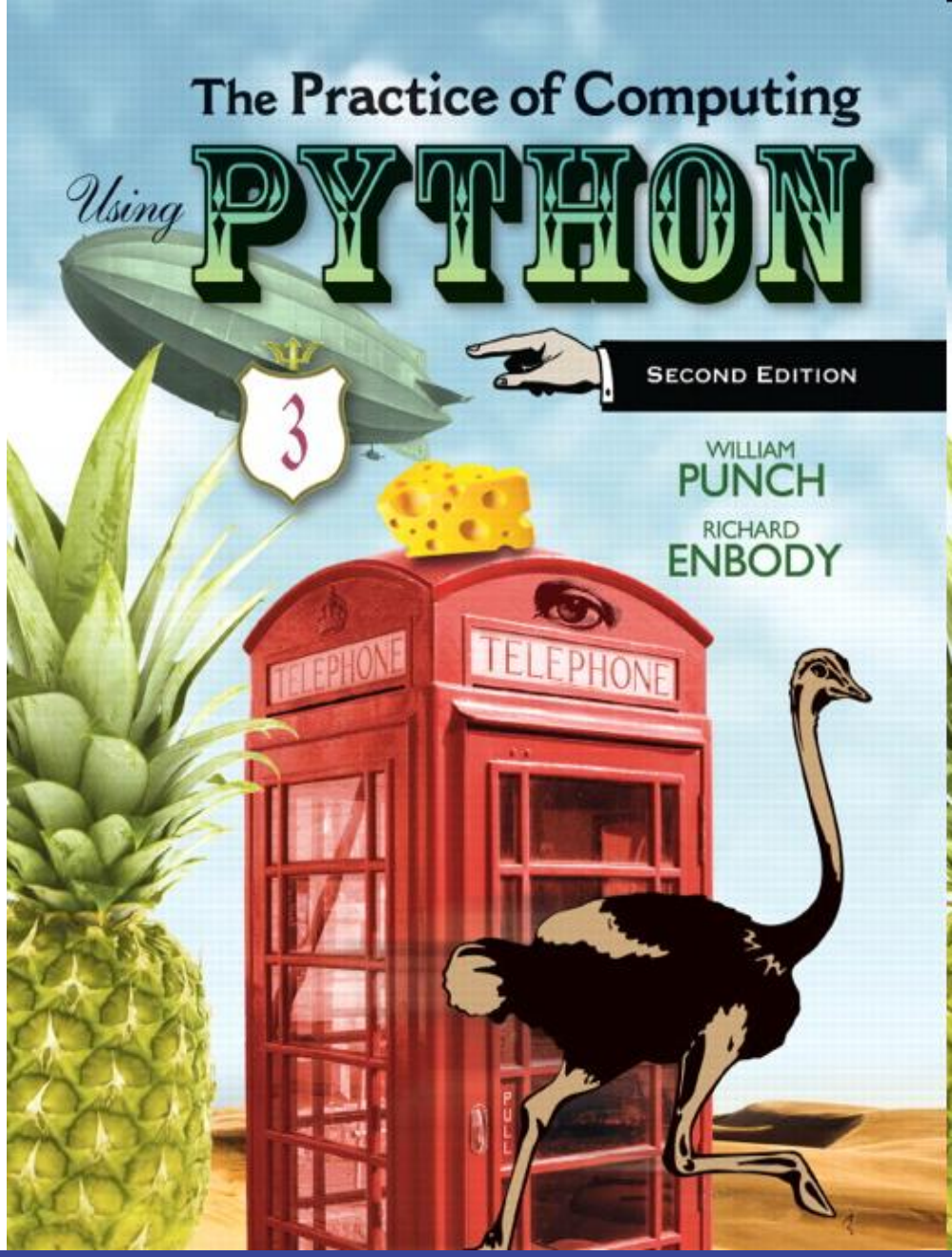
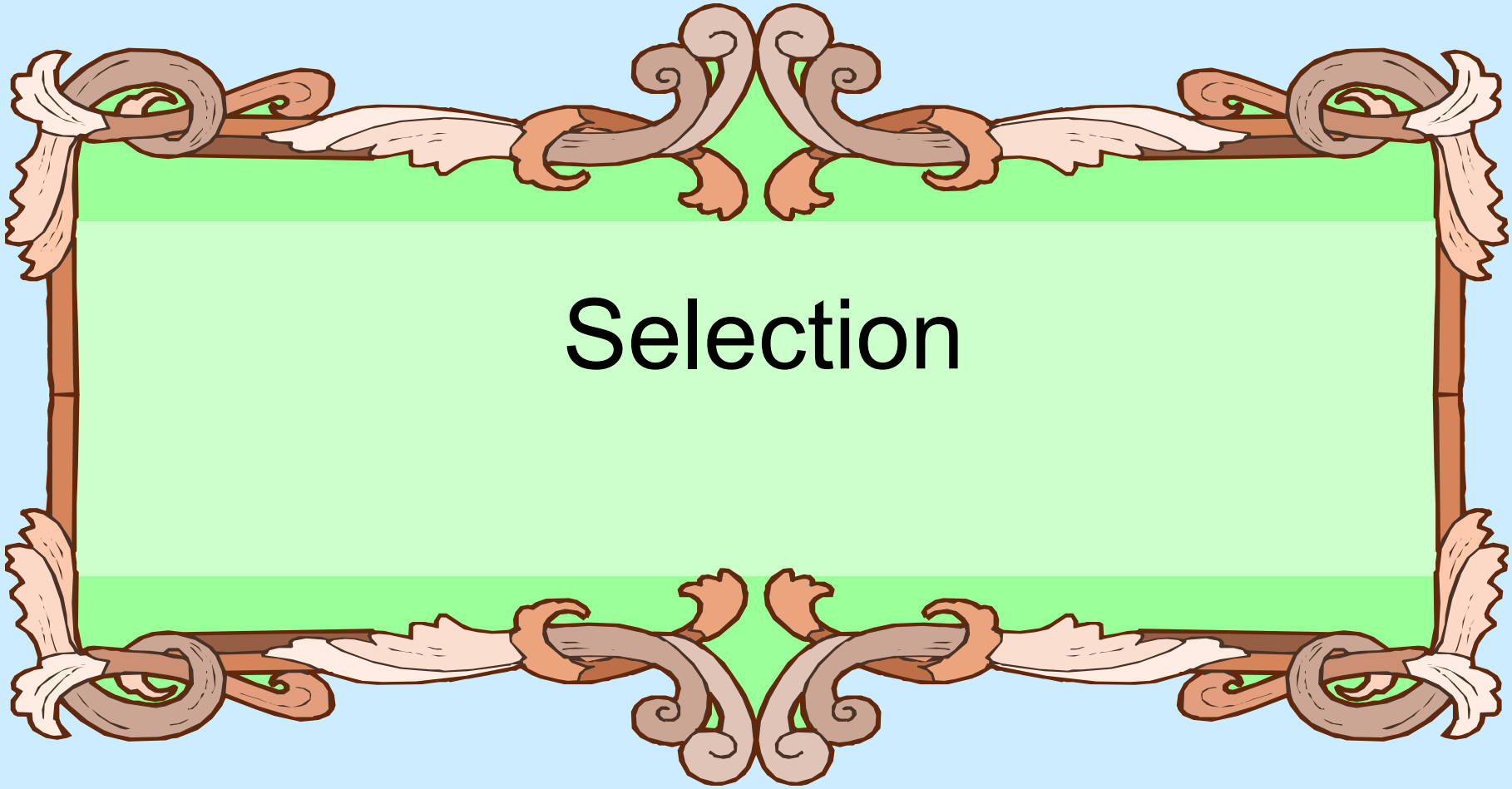
Chapter 2

Control



Control, Quick Overview



Selection

- Selection is how programs make choices, and it is the process of making choices that provides a lot of the power of computing

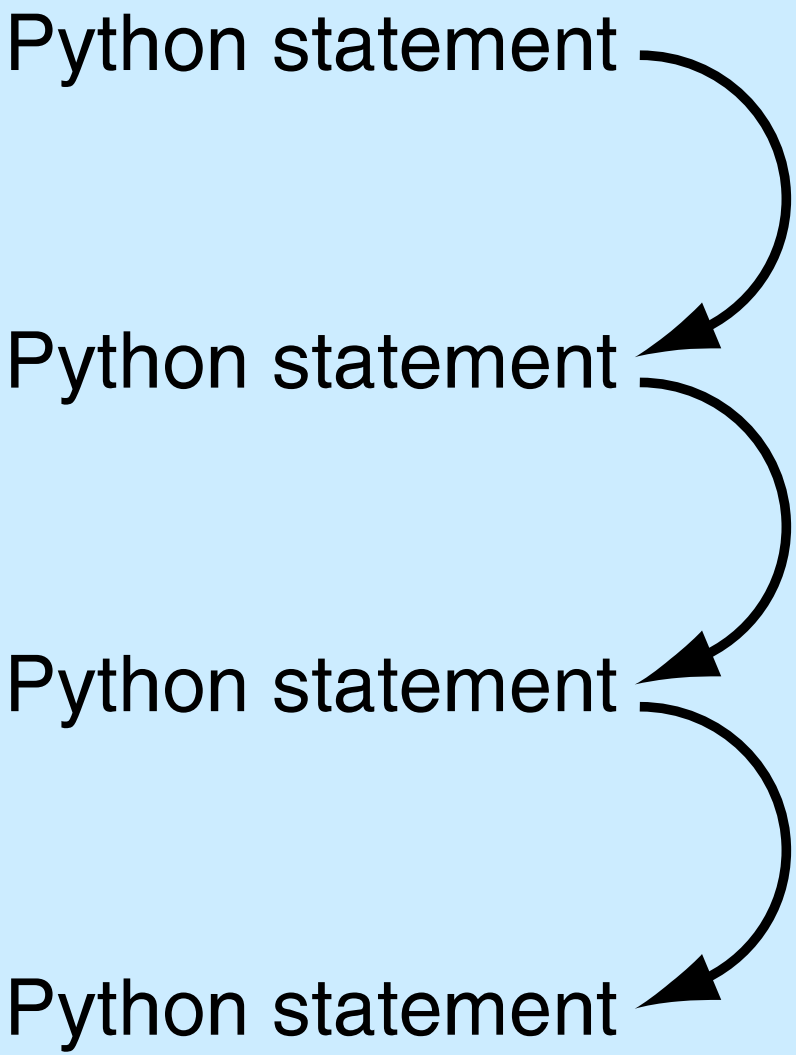


FIGURE 2.1 Sequential program flow.

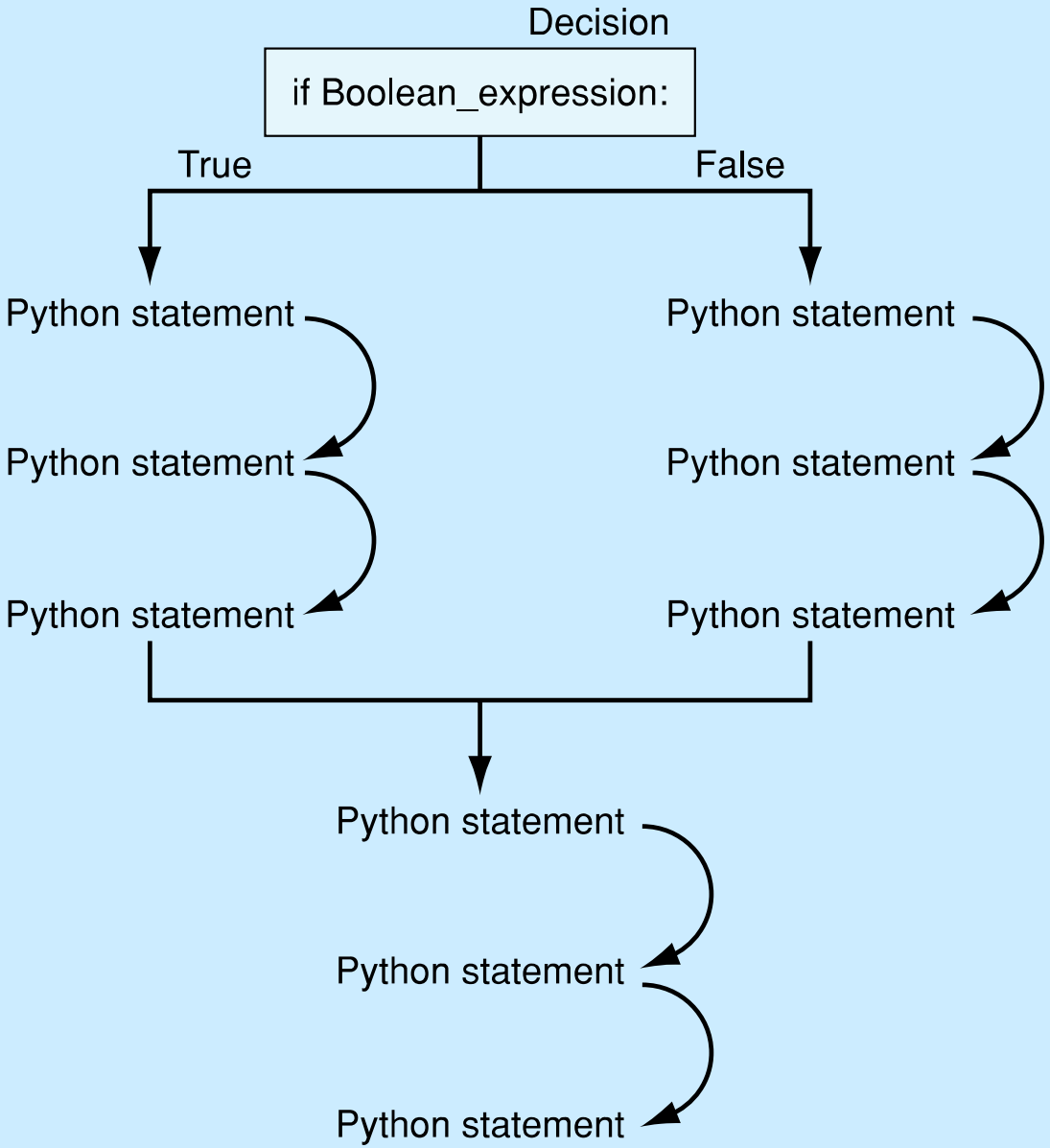


FIGURE 2.2 Decision making flow of control.

<	less than
>	greater than
<=	less than or equal to
>=	greater than or equal to
==	equal to
!=	not equal to

TABLE 2.1 Boolean Operators.

Note that == is equality,
= is assignment

Python if statement

```
if boolean expression :
    suite
```

- evaluate the boolean (`True` or `False`)
- if `True`, execute all statements in the suite

Warning about indentation

- Elements of the suite must all be indented the same number of spaces/tabs
- Python only recognizes suites when they are indented the same distance (***standard is 4 spaces***)
- You must be careful to get the indentation right to get suites right.

Python Selection, Round 2

```
if boolean expression:
    suite1
else:
    suite2
```

The process is:

- evaluate the boolean
- if `True`, run suite1
- if `False`, run suite2

```
>>> first_int = 10
>>> second_int = 20
>>> if first_int > second_int:
        print("The first int is bigger!")
    else:
        print("The second int is bigger!")


The second int is bigger!
>>>
```

Safe Lead in Basketball

- Algorithm due to Bill James (www.slate.com)
- under what conditions can you safely determine that a lead in a basketball game is insurmountable?

The algorithm

- Take the number of points one team is ahead
- Subtract three
- Add ½ point if team that is ahead has the ball, subtract ½ point otherwise
- Square the result
- If the result is greater than the number of seconds left, the lead is safe

first cut

```
# 3. Add a half-point if the team that is ahead has the ball,
#    and subtract a half-point if the other team has the ball.

has_ball_str = input("Does the lead team have the ball (Yes or No): ")

if has_ball_str == "Yes":
    lead_calculation_float = lead_calculation_float + 0.5
else:
    lead_calculation_float = lead_calculation_float - 0.5
```

Problem, what if the lead is less than 0?

second cut

```
# 3. Add a half-point if the team that is ahead has the ball,
#    and subtract a half-point if the other team has the ball.

has_ball_str = input("Does the lead team have the ball (Yes or No): ")

if has_ball_str == 'Yes':
    lead_calculation_float = lead_calculation_float + 0.5
else:
    lead_calculation_float = lead_calculation_float - 0.5

# (Numbers less than zero become zero)
if lead_calculation_float < 0:
    lead_calculation_float = 0
```

catch the lead less than 0

```
# 1. Take the number of points one team is ahead.
points_str = input("Enter the lead in points: ")
points_remaining_int = int(points_str)

# 2. Subtract three.
lead_calculation_float= float(points_remaining_int - 3)

# 3. Add a half-point if the team that is ahead has the ball,
#    and subtract a half-point if the other team has the ball.
has_ball_str = input("Does the lead team have the ball (Yes or No): ")

if has_ball_str == 'Yes':
    lead_calculation_float= lead_calculation_float + 0.5
else:
    lead_calculation_float= lead_calculation_float - 0.5

# (Numbers less than zero become zero)
if lead_calculation_float< 0:
    lead_calculation_float= 0

# 4. Square that.
lead_calculation_float= lead_calculation_float** 2

# 5. If the result is greater than the number of seconds left in the game,
#    the lead is safe.
seconds_remaining_int = int(input("Enter the number of seconds remaining: "))

if lead_calculation_float> seconds_remaining_int:
    print("Lead is safe.")
else:
    print("Lead is not safe.")
```

Repetition, quick overview

Repeating statements

- Besides selecting which statements to execute, a fundamental need in a program is repetition
  - repeat a set of statements under some conditions
- With both selection and repetition, we have the two most necessary programming statements

While and For statements

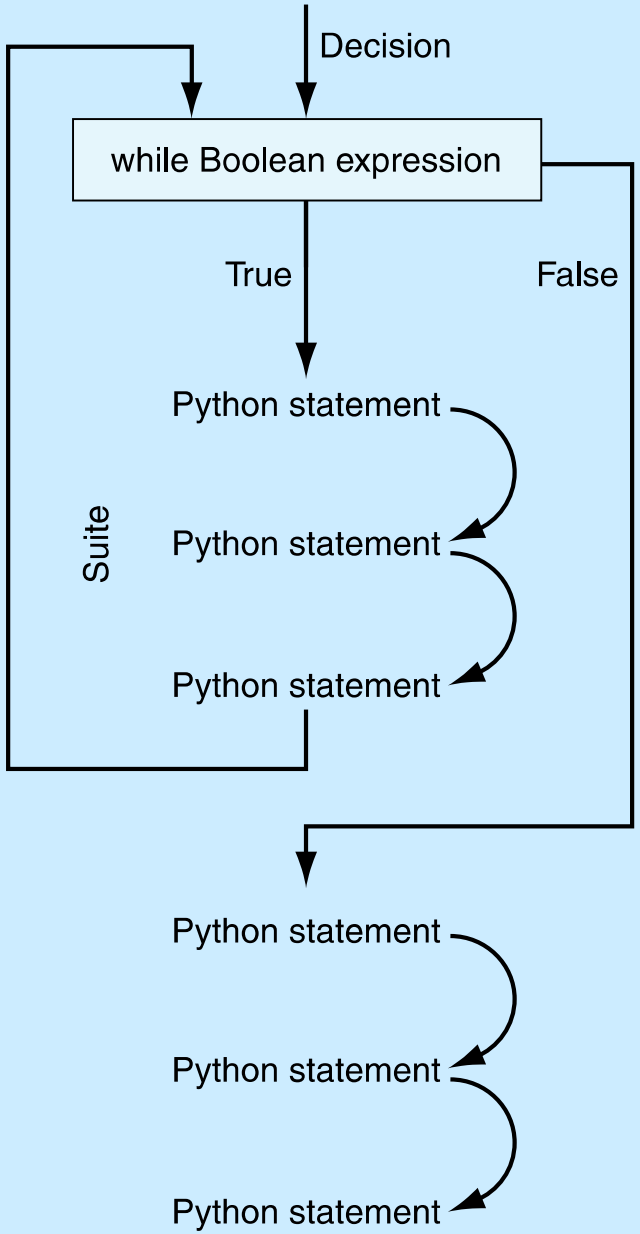
- The `while` statement is the more general repetition construct. It repeats a set of statements while some condition is True.
- The `for` statement is useful for iteration, moving through all the elements of data structure, one at a time.

`while` loop

- Top-tested loop (pretest)
  - test the boolean before running
  - test the boolean before each iteration of the loop

```
while boolean expression:
    suite
```

FIGURE 2.4 ***while*** loop.



repeat while the boolean is true

- while loop will repeat the statements in the suite while the boolean is `True` (or its Python equivalent)
- If the Boolean expression never changes during the course of the loop, the loop will continue forever.

```
1  # simple while
2
3  x_int = 0      # initialize loop-control variable
4
5  # test loop-control variable at beginning of loop
6  while x_int < 10:
7      print(x_int, end=' ')  # print the value of x_int each time through the
                                while loop
8      x_int = x_int + 1      # change loop-control variable
9
10 print()
11 print("Final value of x_int: ", x_int)  # bigger than value printed in loop!
```

General approach to a while

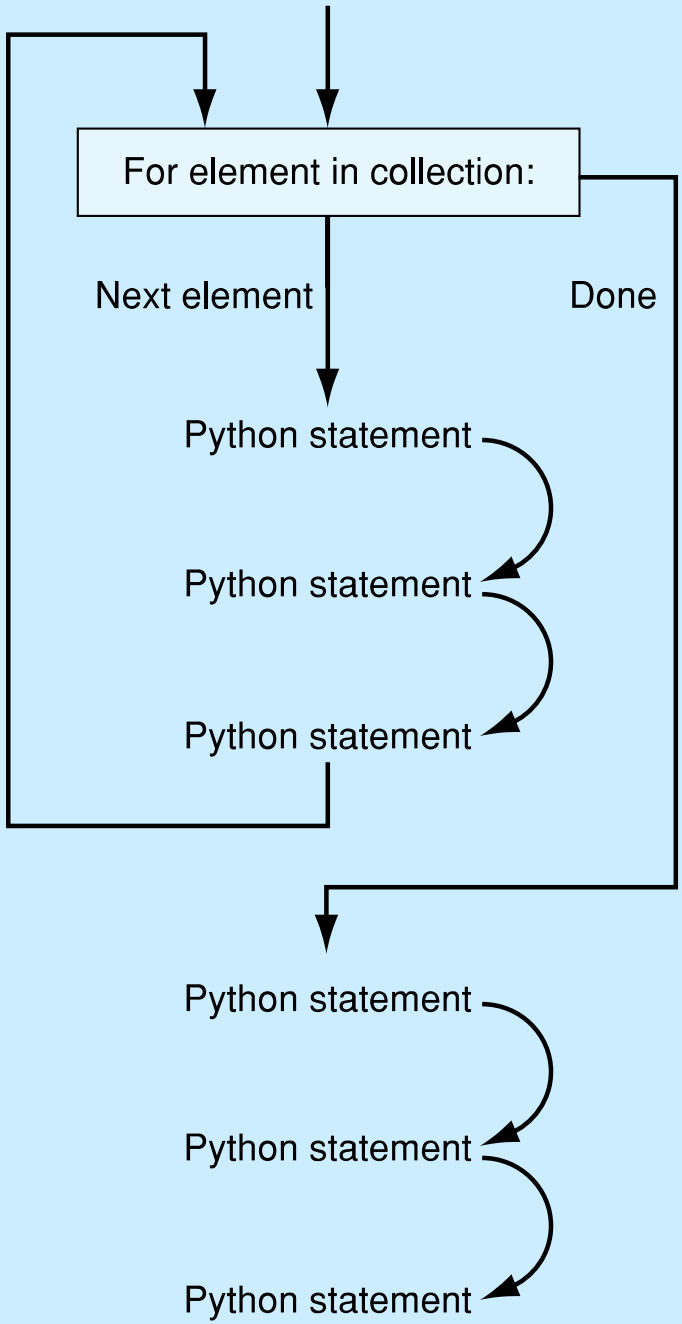
- outside the loop, initialize the boolean
- somewhere inside the loop you perform some operation which changes the state of the program, eventually leading to a False boolean and exiting the loop
- Have to have both!

for and iteration

- One of Python's strength's is it's rich set of built-in data structures
- The for statement iterates through each element of a collection (list, etc.)

```
for element in collection:
    suite
```

FIGURE 2.5 Operation of a ***for*** loop.



Perfect Number Example

a perfect number

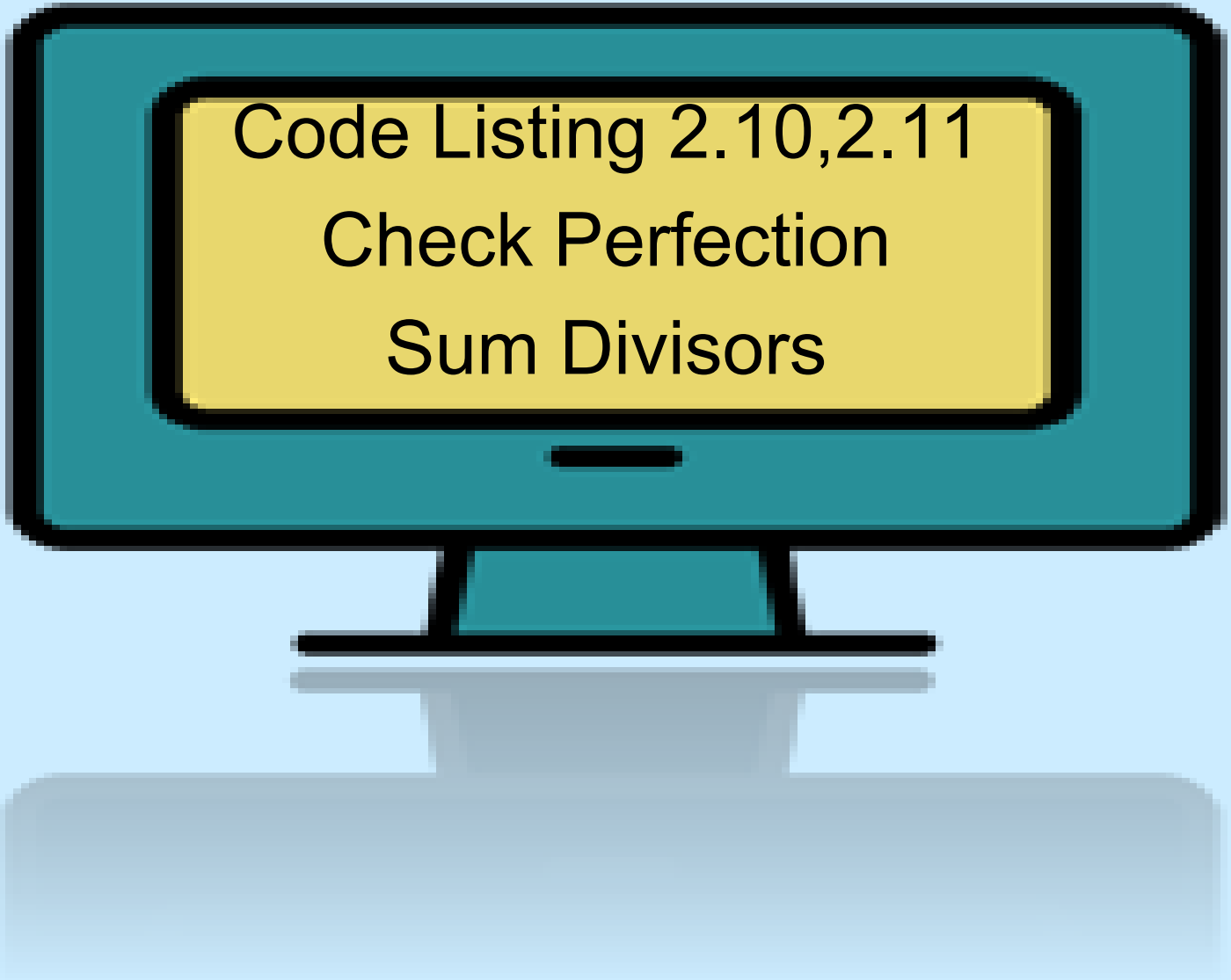
- numbers and their factors were mysterious to the Greeks and early mathematicians
- They were curious about the properties of numbers as they held some significance
- A perfect number is a number whose sum of factors (excluding the number) equals the number
- First perfect number is: 6 (1+2+3)

abundant, deficient

- abundant numbers summed to more than the number.
  - 12: 1+2+3+4+6 =16
- deficient numbers summed to less than the number.
  - 13: 1

design

- prompt for a number
- for the number, collect all the factors
- once collected, sum up the factors
- compare the sum and the number and respond accordingly



Code Listing 2.10

```
if number_int == sum_of_divisors_int:
    print(number_int,"is perfect")
else:
    print(number_int,"is not perfect")
```

Code Listing 2.11

```
divisor = 1
sum_of_divisors = 0
while divisor < number:
    if number % divisor == 0:          # divisor evenly divides theNum
        sum_of_divisors = sum_of_divisors + divisor
    divisor = divisor + 1
```

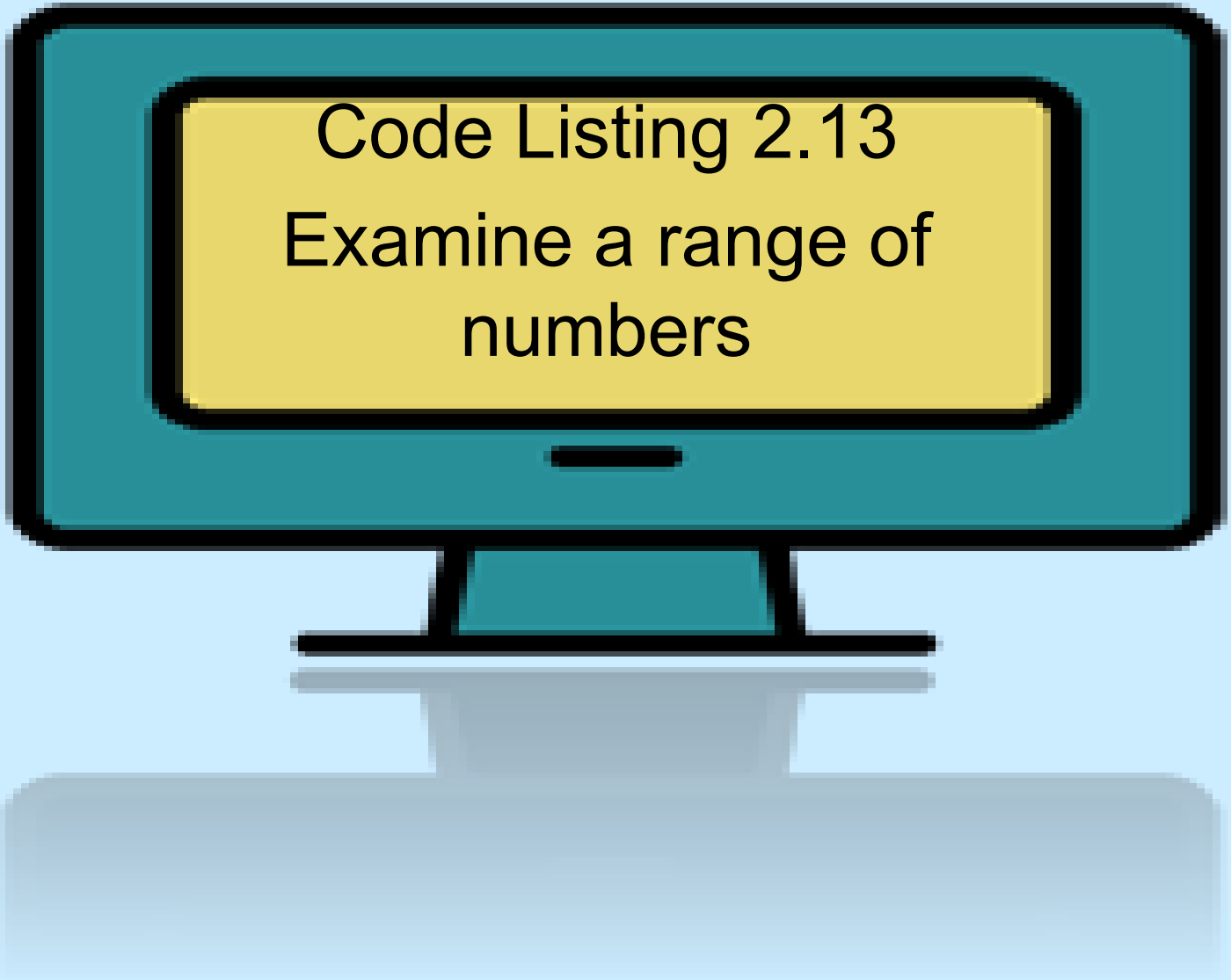
Improving the Perfect Number Program

Work with a range of numbers

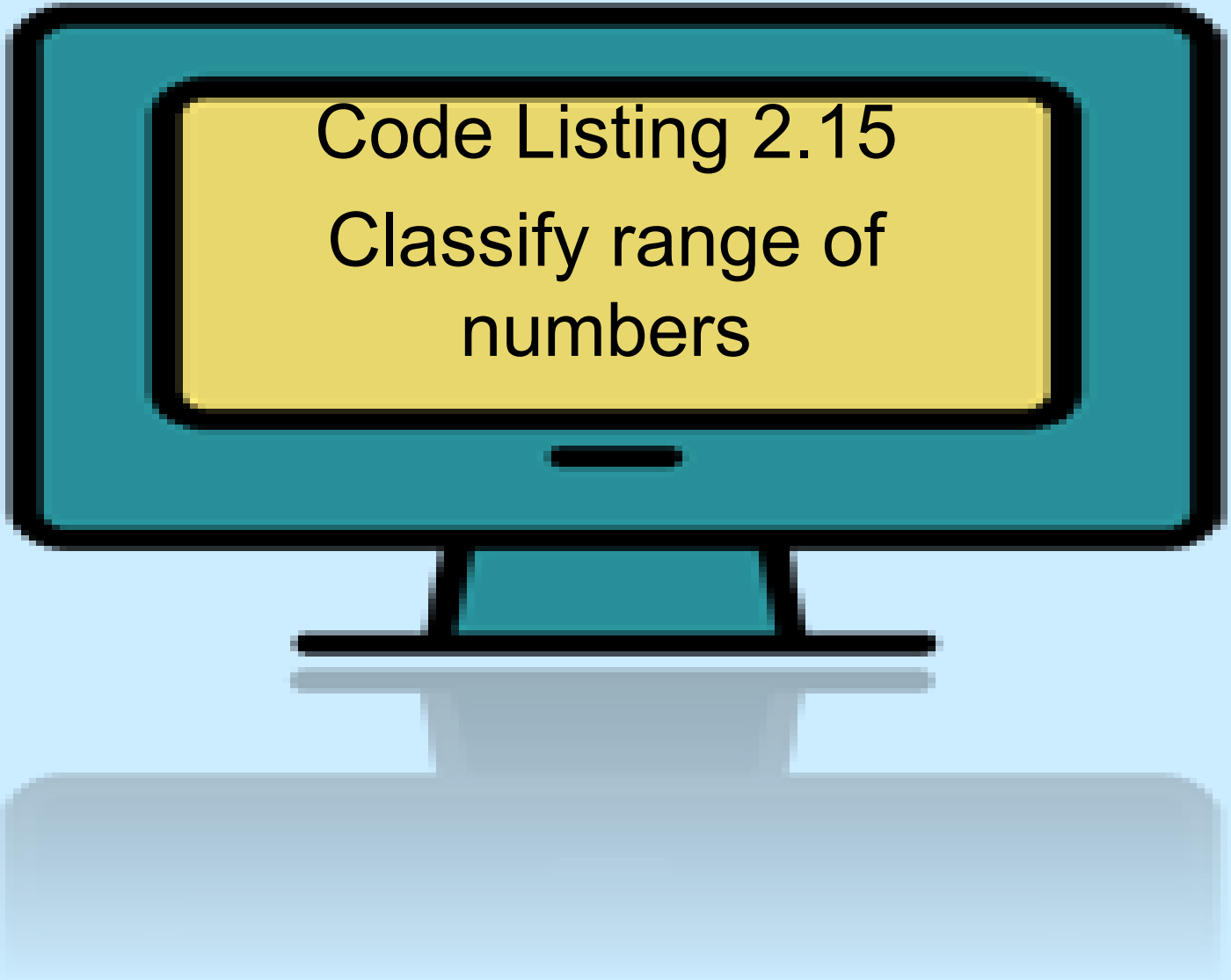
For each number in the range of numbers:

- collect all the factors
- once collected, sum up the factors
- compare the sum and the number and respond accordingly

Print a summary



```
top_num_str = input("What is the upper number for the range:")
top_num = int(top_num_str)
number=2
while number <= top_num:
    # sum the divisors of number
    # classify the number based on its divisor sum
    number += 1
```

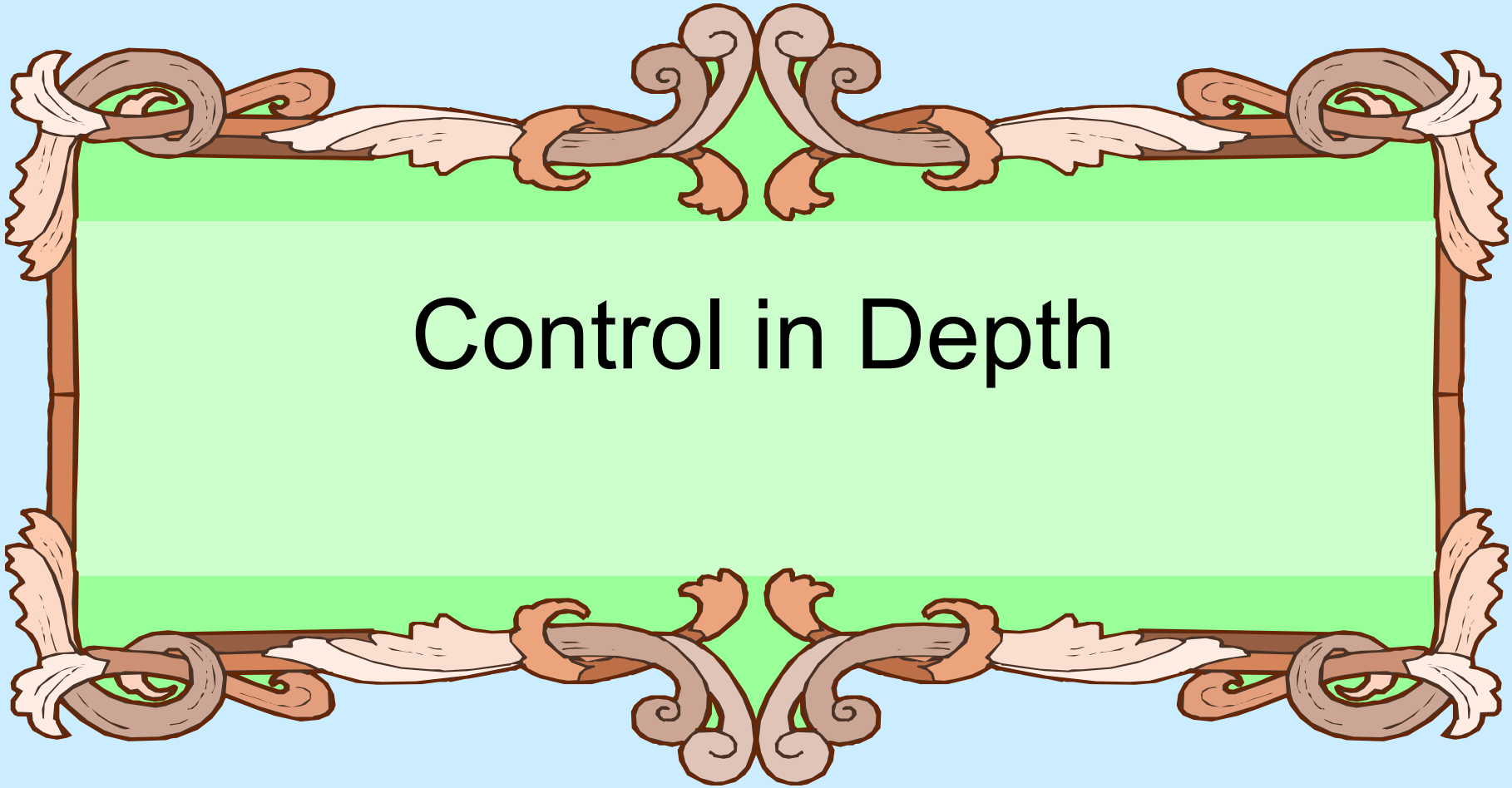


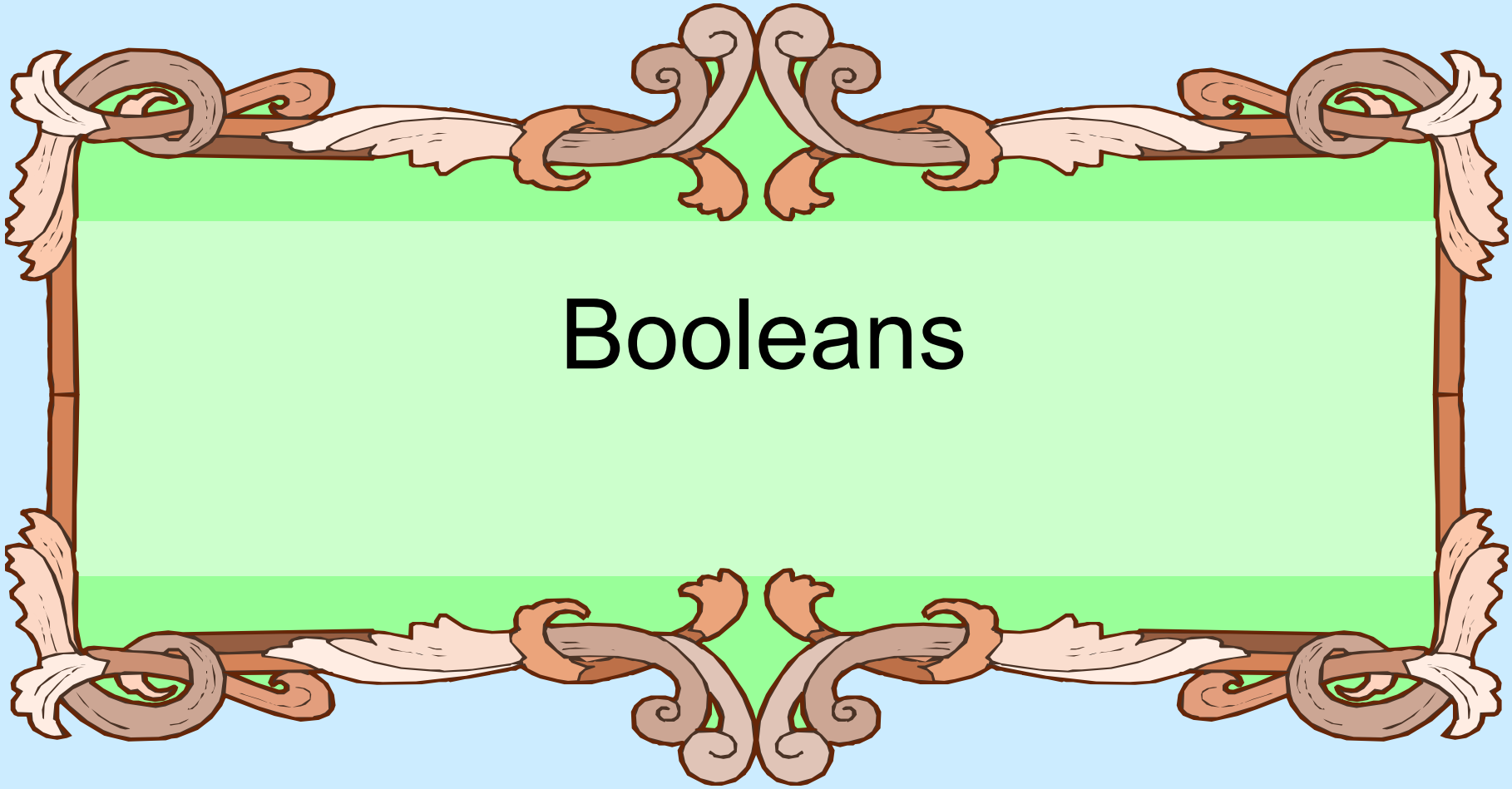
Code Listing 2.15

```
# classify a range of numbers with respect to perfect, adundant or deficient
# unless otherwise stated, variables are assumed to be of type int. Rule 4

top_num_str = input("What is the upper number for the range:")
top_num = int(top_num_str)
number=2
while number <= top_num:
    # sum up the divisors
    divisor = 1
    sum_of_divisors = 0
    while divisor < number:
        if number % divisor == 0:
            sum_of_divisors = sum_of_divisors + divisor
        divisor = divisor + 1

    # classify the number based on its divisor sum
    if number == sum_of_divisors:
        print(number,"is perfect")
    if number < sum_of_divisors:
        print(number,"is abundant")
    if number > sum_of_divisors:
        print(number,"is deficient")
    number += 1
```





Boolean Expressions

- George Boole's (mid-1800's) mathematics of logical expressions
- Boolean expressions (conditions) have a value of True or False
- Conditions are the basis of choices in a computer, and, hence, are the basis of the appearance of intelligence in them.

What is True, and what is False

- true: any nonzero number or nonempty object. `1, 100, "hello", [a,b]`
- false: a zero number or empty object. `0, "",[ ]`
- Special values called `True` and `False`, which are just subs for 1 and 0. However, they print nicely (`True` or `False`)
- Also a special value, `None`, less than everything and equal to nothing

Boolean expression

- Every boolean expression has the form:
  - expression booleanOperator expression
- The result of evaluating something like the above is also just true or false.
- However, remember what constitutes true or false in Python!

Relational Operators

- `3 > 2` ➔ `True`
- Relational Operators have low preference
  - `5 + 3 < 3 - 2`
  - `8 < 1` ➔ `False`
- `'1' < 2` ➔ Error
  - can only compare like types
- `int('1') < 2` ➔ `True`
  - like types, regular compare

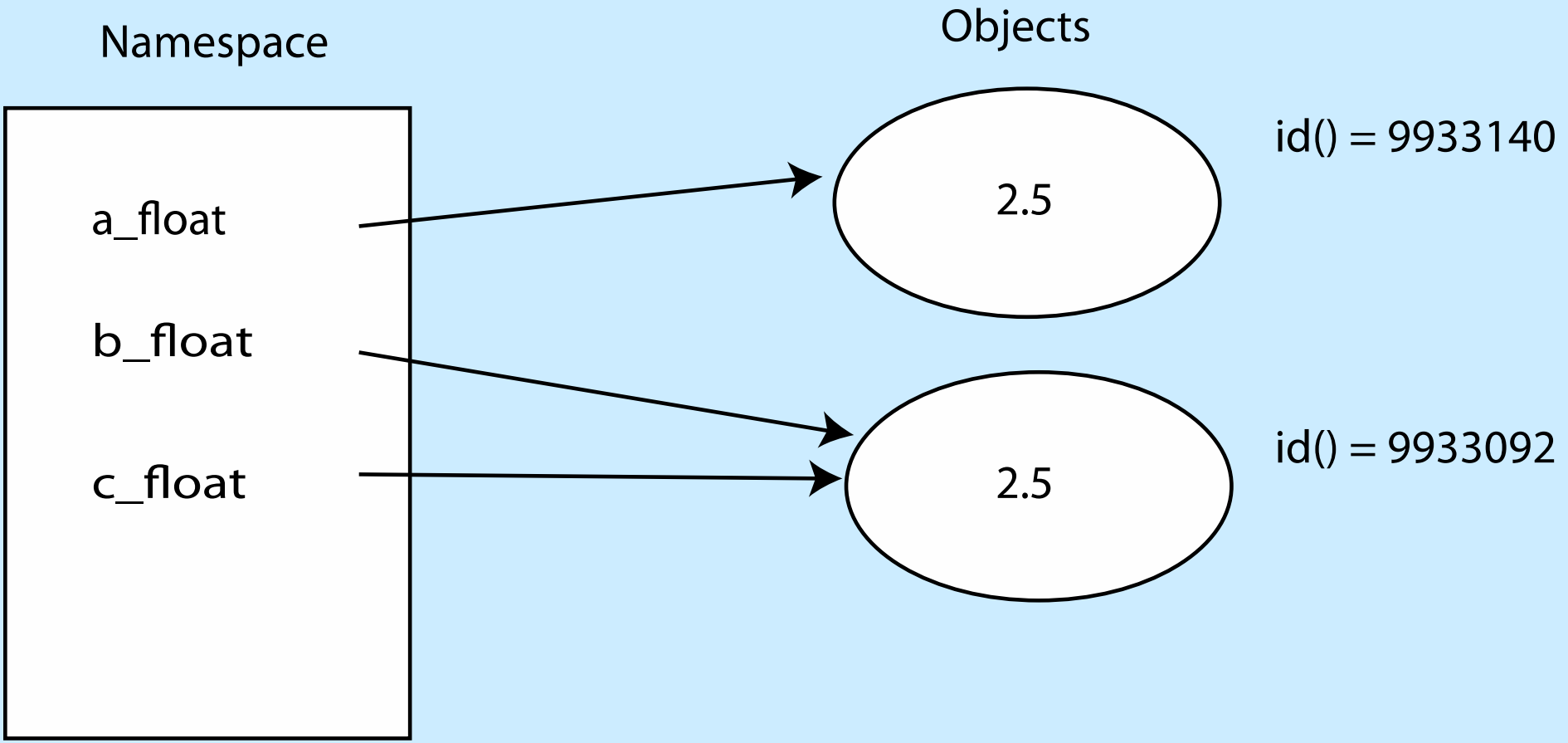
What does Equality mean?

Two senses of equality

- two variables refer to different objects, each object representing the same value
- two variables refer to the same object. The `id()` function used for this.

FIGURE 2.6 What is equality?

a_float = 2.5
b_float = 2.5
c_float = b_float



equal vs. same

- == compares values of two variable's objects, do they represent the same value
- is operator determines if two variables are associated with the same value

From the figure:

```
a_float == b_float → True
a_float is b_float → False
b_float is c_float → True
```

Chained comparisons

- In Python, chained comparisons work just like you would expect in a mathematical expression:
- Given myInt has the value 5
  - `0 <= myInt <= 5` → `True`
  - `0 < myInt <= 5 > 1` → `False`

Pitfall

floating point arithmetic is approximate!

```
>>> u = 11111113
>>> v = -11111111
>>> w = 7.51111111
>>> (u + v) + w
9.51111111
>>> u + (v + w)
9.511111110448837
>>> (u + v) + w == u + (v + w)
False
```

compare using "close enough"

Establish a level of "close enough" for equality

```
>>> u = 11111113
>>> v = -11111111
>>> w = 7.51111111
>>> x = (u + v) + w
>>> y = u + (v + w)
>>> x == y
False
>>> abs(x - y) < 0.0000001  # abs is absolute value
True
```

Compound Expressions

Python allows bracketing of a value between two Booleans, as in math

```
a_int = 5
0 <= a_int <= 10   → True
```

- `a_int >= 0` and `a_int <= 10`
- `and`, `or`, `not` are the three Boolean operators in Python

Truth Tables

p	q	not p	p and q	p or q
True	True			
True	False			
False	True			
False	False			

Truth Tables

p	q	not p	p and q	p or q
True	True	False		
True	False	False		
False	True	True		
False	False	True		

Truth Tables

p	q	not p	p and q	p or q
True	True		True	
True	False		False	
False	True		False	
False	False		False	

Truth Tables

p	q	not p	p and q	p or q
True	True			True
True	False			True
False	True			True
False	False			False

Truth Tables

p	q	not p	p and q	p or q
True	True	False	True	True
True	False	False	False	True
False	True	True	False	True
False	False	True	False	False

Compound Evaluation

- Logically `0 < a_int < 3` is actually `(0 < a_int) and (a_int < 3)`
- Evaluate using `a_int` with a value of 5: `(0< a_int) and (a_int < 3)`
- Parenthesis first: `(True) and (False)`
- Final value: `False`

- (Note: parenthesis are not necessary in this case.)

Precedence & Associativity

Relational operators have precedence and associativity just like numerical operators.

Operator	*Description*
()	Parenthesis (grouping)
**	Exponentiation
+x, -x	Positive, Negative
*,/,%	Multiplication, Division, Remainder
+,-	Addition, Subtraction
<, <=, >, >=,! =, ==	Comparisons
not x	Boolean NOT
and	Boolean AND
or	Boolean OR

TABLE 2.2 Precedence of Relational and Arithmetic Operators: Highest to Lowest

Boolean operators vs. relationals

- Relational operations always return `True` or `False`
- Boolean operators (`and, or`) are different in that:
  - They can return values (that represent `True` or `False`)
  - They have ***short circuiting***

Remember!

- `0, '',[ ]` or other “empty” objects are equivalent to `False`
- anything else is equivalent to `True`

Ego Search on Google

- Google search uses Booleans
- by default, all terms are and'ed together
- you can specify or (using OR)
- you can specify not (using -)
- Example is:

```
'Punch' and ('Bill' or 'William') and not
'gates'
```

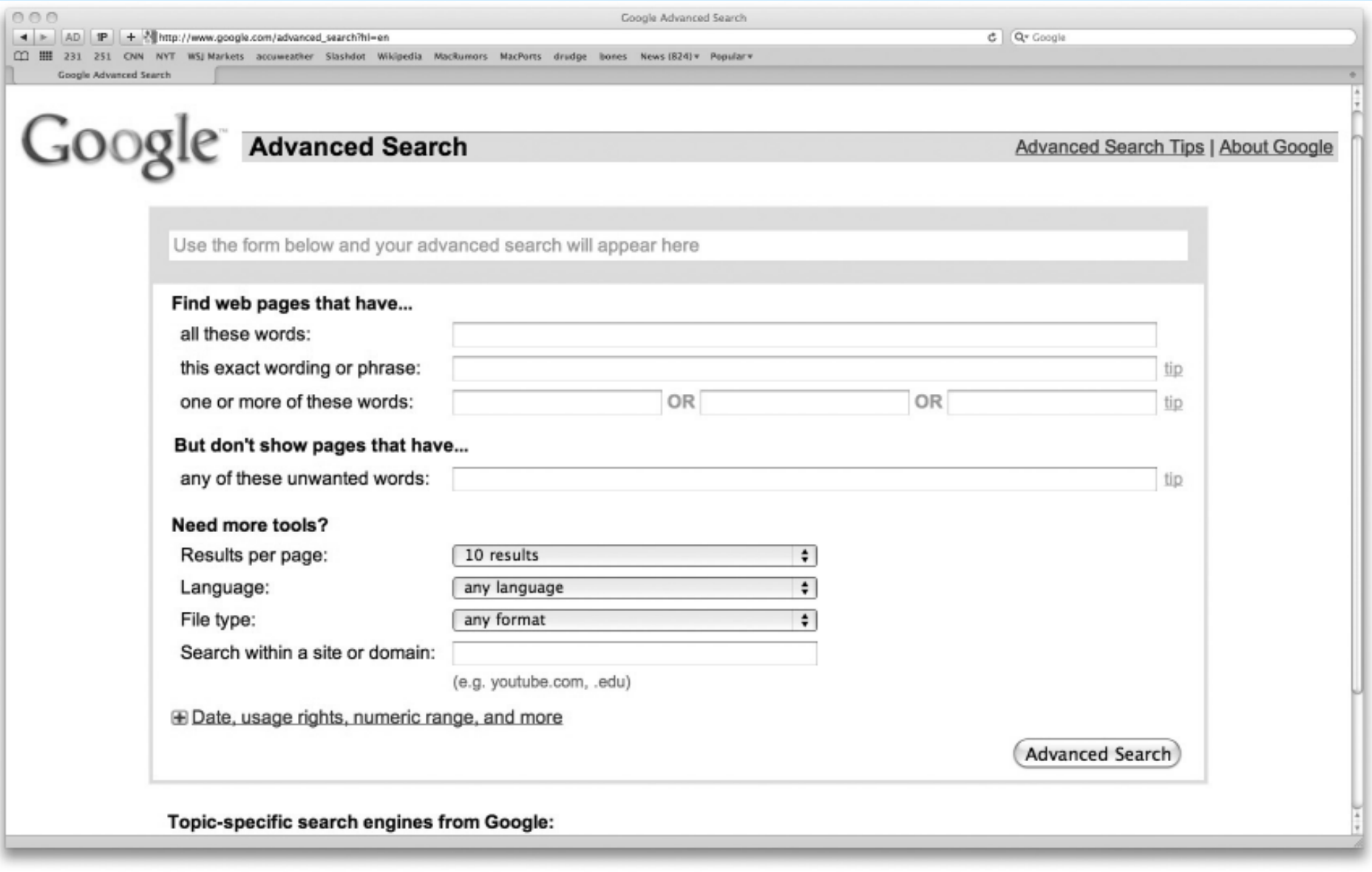


FIGURE 2.7 The Google advanced search page.

More on Assignments

Remember Assignments?

- Format: `lhs = rhs`
- Behavior:
  - expression in the rhs is evaluated producing a value
  - the value produced is placed in the location indicated on the lhs

Can do multiple assignments

```
a_int, b_int = 2, 3
```

first on right assigned to first on left, second on right assigned to second on left

```
print(a_int, b_int)   # prints 2 3
```

```
a_int,b_int = 1,2,3
```
➔ Error

counts on lhs and rhs must match

traditional swap

- Initial values: `a_int= 2, b_int = 3`
- Behavior: swap values of `X` and `Y`
  - Note: `a_int = b_int`

    `a_int = b_int` doesn't work (why?)
  - introduce extra variable `temp`
    - `temp = a_int` # save a_int value in temp
    - `a_int = b_int` # assign a_int value to b_int
    - `b_int = temp` # assign temp value to b_int

Swap using multiple assignment

```
a_int, b_int = 2, 3
print(a_int, b_int)  # prints 2  3

a_int, b_int = b_int, a_int
print(a_int, b_int)  # prints 3  2
```

remember, evaluate all the values on the rhs first, then assign to variables on the lhs

Chaining for assignment

Unlike other operations which chain left to right, assignment chains right to left

```
a_int = b_int = 5
print(a_int, b_int)  # prints 5  5
```

More Control: Selection

Compound Statements

- Compound statements involve a set of statements being used as a group
- Most compound statements have:
  - a header, ending with a `:` (colon)
  - a suite of statements to be executed
- `if, for, while` are examples of compound statements

General format, suites

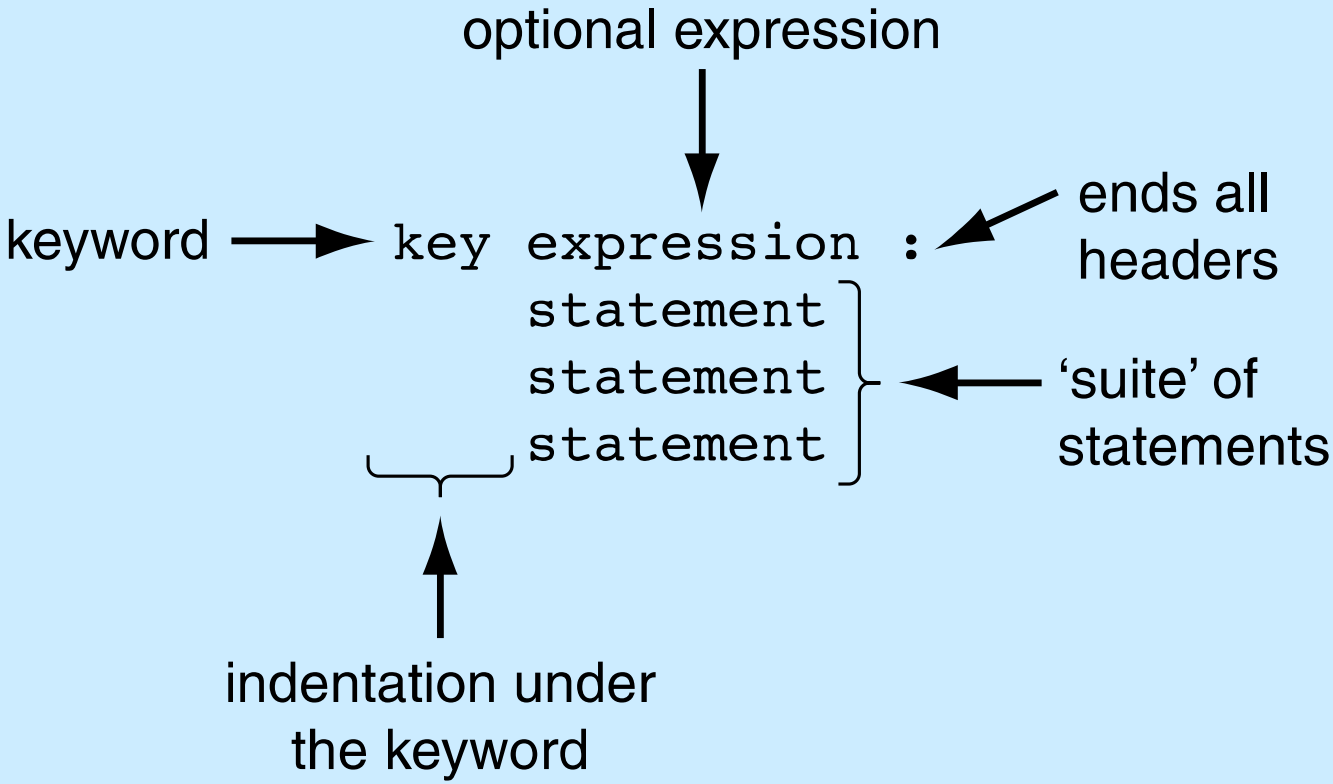


Figure 2.3: Control expression.

Have seen 2 forms of selection

```
if boolean expression:
    suite

if boolean expression:
    suite
else:
    suite
```

Python Selection, Round 3

```
if boolean expression1:
        suite1
elif boolean expression2:
        suite2
(as many elif's as you want)
else:
        suite_last
```

if, elif, else, the process

- evaluate Boolean expressions until:
  - the Boolean expression returns `True`
  - none of the Boolean expressions return `True`
- if a boolean returns `True`, run the corresponding suite. Skip the rest of the `if`
- if no boolean returns `True`, run the `else` suite, the default suite

```
percent_float = float(input("What is your percentage? "))

if 90 <= percent_float < 100:
    print("you received an A")
elif 80 <= percent_float < 90:
    print("you received a B")
elif 70 <= percent_float < 80:
    print("you received a C")
elif 60 <= percent_float < 70:
    print("you received a D")
else:
    print("oops, not good")
```

What happens if `elif` are replaced by `if`?

```
# classify the number based on its divisor sum
if number == sum_of_divisors:
    print(number,"is perfect")
elif number < sum_of_divisors:
    print(number,"is abundant")
else:
    print(number,"is deficient")
number += 1
```

More Control: Repetition

Developing a while loop

Working with the ***loop control variable***:

- Initialize the variable, typically outside of the loop and before the loop begins.
- The condition statement of the while loop involves a Boolean using the variable.
- Modify the value of the control variable during the course of the loop

Issues:

Loop never starts:

- the control variable is not initialized as you thought (or perhaps you don't always want it to start)

Loop never ends:

- the control variable is not modified during the loop (or not modified in a way to make the Boolean come out `False`)

while loop, round two

- while loop, oddly, can have an associated `else` suite
- `else` suite is executed when the loop finishes under normal conditions
  - basically the last thing the loop does as it exits

while with else

```
while booleanExpression:
  suite
  suite
else:
  suite
  suite
rest of the program
```

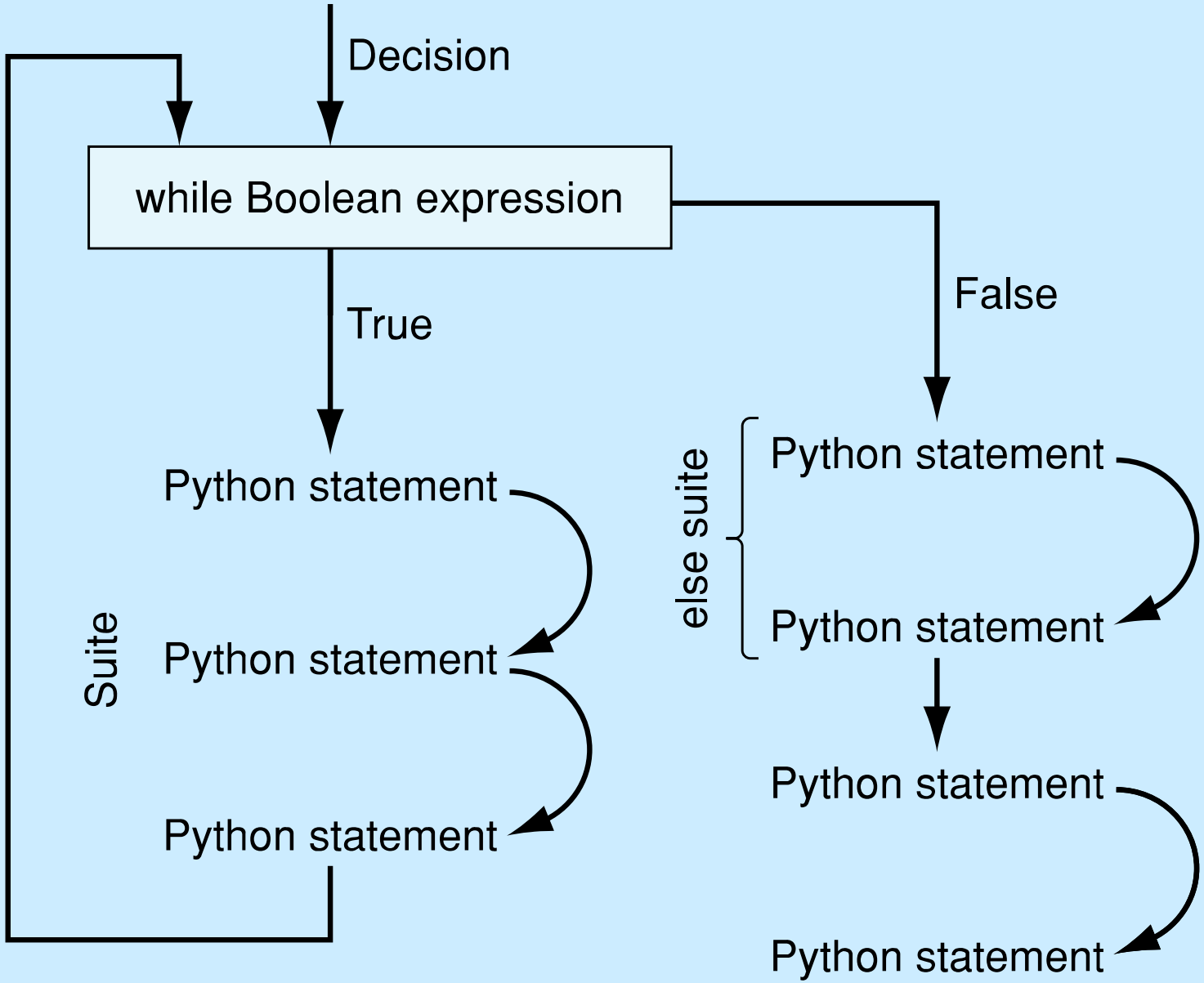


FIGURE 2.9 ***while-else.***

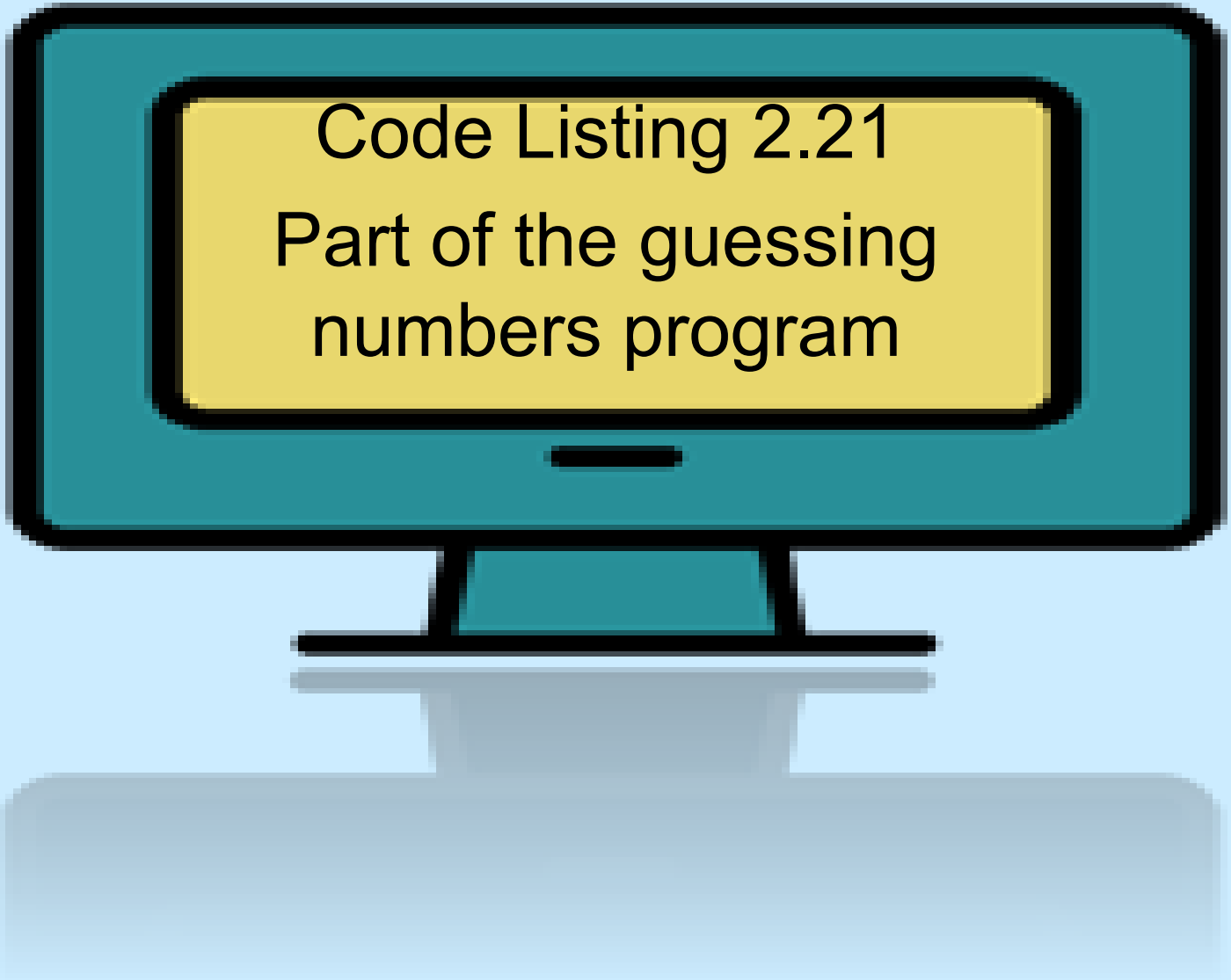
Break statement

- A `break` statement in a loop, if executed, exits the loop
- It exists immediately, skipping whatever remains of the loop as well as the else statement (if it exists) of the loop

```
14 # get an initial guess
15 guess_str = input("Guess a number: ")
16 guess = int(guess_str)  # convert string to number
17
18 # while guess is range, keep asking
19 while 0 <= guess <= 100:
20     if guess > number:
21         print("Guessed Too High.")
22     elif guess < number:
23         print("Guessed Too Low.")
24     else:                    # correct guess, exit with break
25         print("You guessed it. The number was:",number)
26         break
27     # keep going, get the next guess
28     guess_str = input("Guess a number: ")
29     guess = int(guess_str)
30 else:
31     print("You quit early, the number was:",number)
```

Continue statement

- A `continue` statement, if executed in a loop, means to immediately jump back to the top of the loop and re-evaluate the conditional
- Any remaining parts of the loop are skipped for the one iteration when the continue was exectued



```
7  # initialize the input number and the sum
8  number_str = input("Number: ")
9  the_sum = 0
10
11 # Stop if a period (.) is entered.
12 # remember, number_str is a string  until we convert it
13 while number_str != "." :
14     number = int(number_str)
15     if number % 2 == 1:  # number is not even (it is odd)
16         print ("Error, only even numbers please.")
17         number_str = input("Number: ")
18         continue      # if the number is not even, ignore it
19     the_sum += number
20     number_str = input("Number: ")
21
22 print ("The sum is:",the_sum)
```

change in control: Break and Continue

- while loops are easiest read when the conditions of exit are clear
- Excessive use of continue and break within a loop suite make it more difficult to decide when the loop will exit and what parts of the suite will be executed each loop.
- Use them judiciously.

While overview

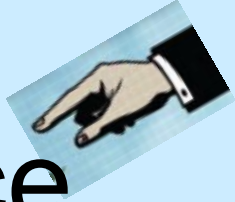
```
while test1:
    statement_list_1
    if test2:  break         # Exit loop now; skip else
    if test3:  continue      # Go to top of loop now
    # more statements
else:
    statement_list_2       # If we didn't hit a 'break'

# 'break' or 'continue' lines can appear anywhere
```

Range and for loop

Range function

- The range function represents a sequence of integers
- the range function takes 3 arguments:
  - the beginning of the range. Assumed to be 0 if not provided
  - the end of the range, but not inclusive (up to but not including the number). Required
  - the step of the range. Assumed to be 1 if not provided
- if only one arg provided, assumed to be the end value



Iterating through the sequence

```
for num in range(1,5):
    print(num)
```

- range represents the sequence 1, 2, 3, 4
- for loop assigns `num` to each of the values in the sequence, one at a time, in sequence
- prints each number (one number per line)

range generates on demand

Range generates its values on demand

```
>>> range(1,10)
range(1, 10)
>>> my_range=range(1,10)
>>> type(my_range)
<class 'range'>
>>> len(my_range)
9
>>> for i in my_range:
        print(i, end=' ')

1 2 3 4 5 6 7 8 9
>>>
```

Hailstone example

Collatz

- The Collatz sequence is a simple algorithm applied to any positive integer
- In general, by applying this algorithm to your starting number you generate a sequence of other positive numbers, ending at 1
- Unproven whether every number ends in 1 (though strong evidence exists)

Algorithm

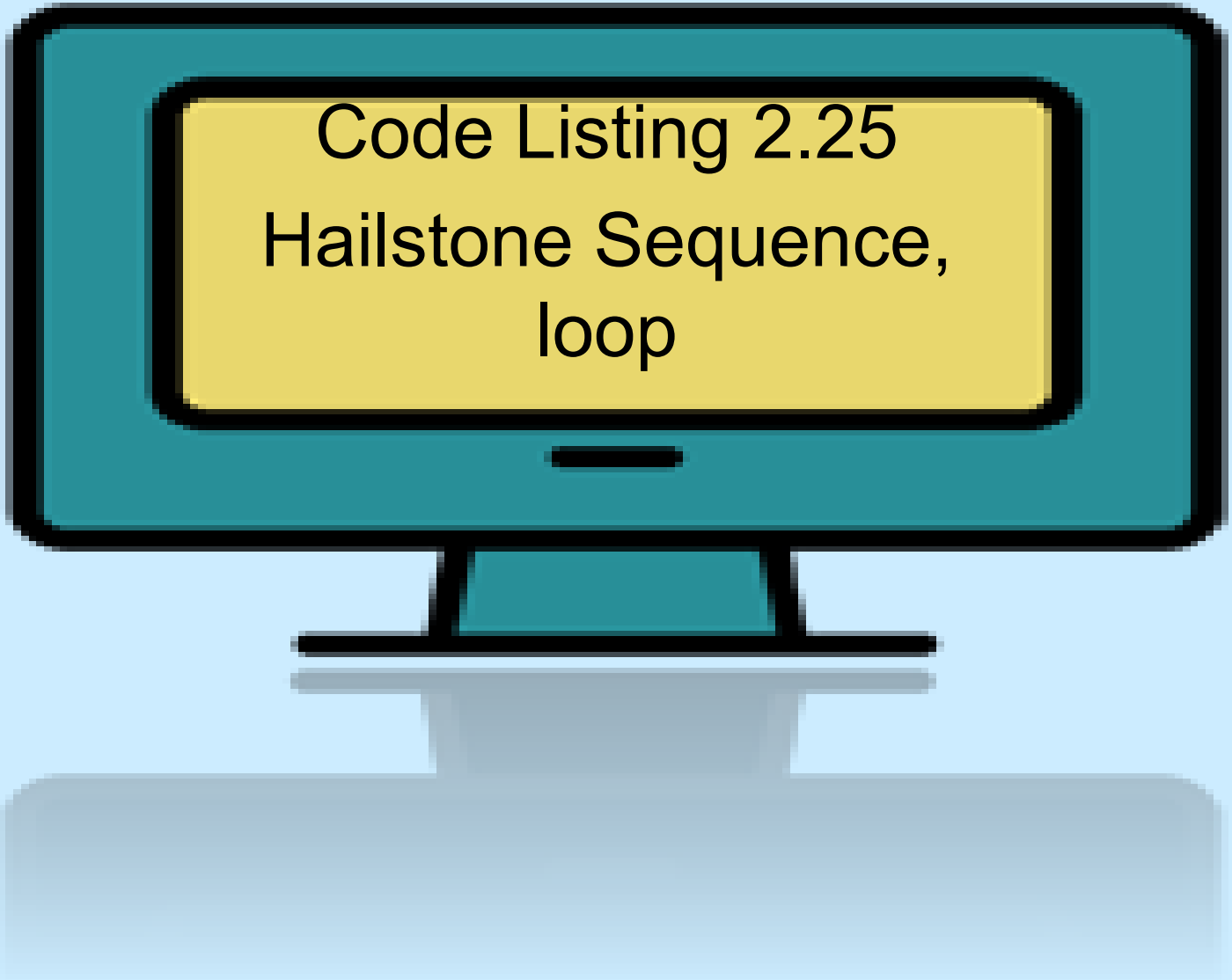
while the number does not equal one

- If the number is odd, multiply by 3 and add 1
- If the number is even, divide by 2
- Use the new number and reapply the algorithm

Even and Odd

Use the remainder operator

- `if num % 2 == 0:` # even
- `if num % 2 == 1:` # odd
- `if num %2:` # odd (why???)



```
1  # Generate a hailstone sequence
2  number_str = input("Enter a positive integer:")
3  number = int(number_str)
4  count = 0
5
6  print("Starting with number:",number)
7  print("Sequence is: ", end=' ')
8
9  while number > 1:  # stop when the sequence reaches 1
10
11     if number%2:          # number is odd
12         number = number*3 + 1
13     else:                 # number is even
14         number = number/2
15     print(number,",", end=' ')    # add number to sequence
16
17     count +=1             # add to the count
18
19 else:
20     print()     # blank line for nicer output
21     print("Sequence is ",count," numbers long")
```

The Rules

1. Think before you program!
2. A program is a human-readable essay on problem solving that also happens to execute on a computer.
3. The best way to imporve your programming and problem solving skills is to practice!
4. A foolish consistency is the hobgoblin of little minds
5. Test your code, often and thoroughly